Calculating modules in contextual logic program refinement

ROBERT COLVIN, IAN J. HAYES and PAUL STROOPER
School of Information Technology and Electrical Engineering,
University of Queensland, Brisbane, Australia
(*e-mail:* [robert,Ian.Hayes,pstroop]@itee.uq.edu.au)

submitted 10 November 2004; revised 24 January 2006; accepted 23 August 2006

To appear in Theory and Practice of Logic Programming (TPLP)

Abstract

The refinement calculus for logic programs is a framework for deriving logic programs from specifications. It is based on a wide-spectrum language that can express both specifications and code, and a refinement relation that models the notion of correct implementation. In this paper we extend and generalise earlier work on *contextual refinement*. Contextual refinement simplifies the refinement process by abstractly capturing the context of a subcomponent of a program, which typically includes information about the values of the free variables. This paper also extends and generalises *module refinement*. A *module* is a collection of procedures that operate on a common data type; module refinement between a specification module A and an implementation module C allows calls to the procedures of A to be systematically replaced with calls to the corresponding procedures of C. Based on the conditions for module refinement, we present a method for *calculating* an implementation module from a specification module. Both contextual and module refinement within the refinement calculus have been generalised from earlier work and the results are presented in a unified framework.
KEYWORDS: Logic programs, refinement, modules, context

1 Introduction

The construction of programs that are correct with respect to their specifications is an important goal of software development. A *refinement calculus* is a formal method for deriving programs from specifications in a step-wise fashion. It is based on:

- a *wide-spectrum language* that can express both specifications and executable programs;
- a *refinement relation* that models the notion of correct implementation; and
- a collection of *refinement laws* providing the means to refine specifications to code in a stepwise fashion.

The wide-spectrum language contains both specification and implementation constructs, blurring the distinction between specifications and executable code. A series

of correctness-preserving refinement laws are applied to a specification, replacing specification constructs with implementation constructs. Each refinement law is proved with respect to the underlying semantics of the calculus. A law may have associated *proof obligations*, which must be discharged to ensure the application of the law is valid.

A refinement calculus for logic programs has been developed (Hayes et al. 1997; Hayes et al. 2000; Hayes et al. 2002). In this paper we extend and generalise earlier work on *contextual* and *module* refinement of logic programs within the refinement calculus, and present the results in a unified framework.

Because our wide-spectrum language is monotonic with respect to the refinement ordering, a program, S, is refined by refining any of its components. We can use this property to decompose the refinement of a program into the refinement of (some or all of) its components. In many situations a component of S may inherit context from S. This context can, for example, provide information about the values of free variables in the component. In this paper we provide a framework for making context available during the refinement of a program's components.

We use contextual refinement to reason about module refinement. A module in our language is a group of procedures that operate on a common data type. By making assumptions about the structure of a program that uses the module we derive a context in which efficient implementations of abstract data types are allowed. Finally, we present a method for deriving, or *calculating*, an implementation module from an abstract module. Starting from the abstract module and a *coupling invariant* — a relation between the abstract and implementation types — a specification of the implementation module can be automatically produced (subject to some consistency checks).

The paper is structured as follows. In Sect. 2 the meaning of wide-spectrum language constructs and refinement are informally described. Sect. 3 examines contextual refinement of logic programs. The contextual refinement laws are illustrated with an example of a data refinement. In Sect. 4 we discuss module refinement, where we reason about groups of procedures that operate on a common data type. In Sect. 5 we present a general scheme for deriving an implementation of a module based on the relationship between the specification and implementation types. We then specialise the scheme for particular combinations of abstract operations and coupling invariants. In particular, Sect. 6 extends the specification language so that nondeterminism in some coupling invariants can be eliminated, allowing more efficient implementation modules. In Sect. 7 we discuss related work.

This paper summarises and extends the first author's thesis (Colvin 2002). We combine and extend the results of earlier papers (Colvin et al. 1998; Colvin et al. 2000; Colvin et al. 2001) and adopt a consistent structure and notation, resulting in a simpler and more comprehensive theory for contextual and data refinement. Specifically, the results of Colvin et al. (2000) are generalised by unifying the treatment of context for the different constructs in the language (Sect. 3), and the results of Colvin et al. (1998) are condensed and simplified in the unified notation (Sect. 3.2). The results of Colvin et al. (2001) are extended by considering more program structures and allowing arbitrary predicates as context, and a more complex example is

used to present the results (Sect. 4). We also present a technique for automatically calculating implementation modules (Sect. 5), originally proposed in Colvin (2002). Specialisations of the calculation technique (Sect. 5.3) and the use of demonic nondeterminism in module calculations (Sect. 6) are novel to this paper.

2 The wide-spectrum language and refinement

A wide-spectrum language may be used to express both specifications as well as executable programs (Partsch 1990). For example, Back (1988) included specification constructs in Dijkstra's imperative language (Dijkstra 1976). Using a wide-spectrum language has the benefit of allowing stepwise refinement within a single notational framework.

2.1 Basic constructs

Semantic model. For brevity we present an informal, intuitive description of the semantics of the language and refinement, and present the main theorems and results as high-level refinement laws. The details of a predicate-based semantics appears in (Hayes et al. 1997), and of an operational semantics in (Hayes et al. 2002).

In our language, a *command* (logic program fragment) S with free variables V constrains (instantiates) V to satisfy S. (This is the same principal involved as when a procedure call $p(V)$ constrains V to satisfy p.) The instantiation of the free variables, which may already be partially or fully instantiated, is the "effect" of S, similar to a postcondition in Hoare logic. Additionally, every command may have an associated "assumption", similar to preconditions in Hoare logic. Assumptions specify the instantiations of the free variables for which the command is guaranteed to function correctly. If the free variables do not satisfy the assumptions, the program may behave in any manner (like **abort** in Dijkstra's language).

The commands in our wide-spectrum language are discussed below (a summary appears in Fig. 1). We describe them in terms of their assumptions (input instantiations) and effect (output instantiations). Throughout the paper we adopt the following naming conventions.

$A, B..$	predicates (inside *assumption* commands)
$P, Q..$	predicates (inside *specification* commands)
$S, T..$	commands
V, X, Y	variables
U	terms

Specifications. A specification $\langle P \rangle$ constrains the instantiations of its free variables so that they satisfy predicate P; it is the basic building block of programs in the wide-spectrum language. For example, the specification $\langle X = 5 \vee X = 6 \rangle$ represents the set of instantiations $\{5, 6\}$ for X. We define two special cases of specification

$\langle P \rangle$	-	specification
$\{\mathrm{A}\}$	-	assumption
$(S \lor T)$	-	disjunction
$(S \land T)$	-	parallel conjunction
(S, T)	-	sequential conjunction
$(\exists\ V \bullet S)$	-	existential quantification
$(\forall\ V \bullet S)$	-	universal quantification
$pc(U)$	-	procedure call

Fig. 1. Summary of commands in the wide-spectrum language

commands:

$$\mathbf{fail} \mathrel{\widehat{=}} \langle \mathit{false} \rangle$$
$$\mathbf{true} \mathrel{\widehat{=}} \langle \mathit{true} \rangle$$

The specification **fail** is not satisfied by any instantiation of free variables; it is like Prolog's `fail`. The specification **true** does nothing, i.e., does not constrain the instantiations; it is like Prolog's `true`. Specification commands operate on any input instantiations, that is, their assumption is always *true*.

Assumptions. An assumption $\{A\}$, where A is a predicate, acts as a precondition, and thus restricts the input instantiations. As such, it provides a context for a program fragment. For example, some program S may require that an integer parameter be non-zero, which can be expressed as "$\{X \neq 0\}, S$". If the assumption does not hold, the program may abort. Aborting includes program behaviour such as non-termination and abnormal termination due to exceptions like division by zero, as well as termination with arbitrary results. We define the (worst possible) program **abort**:

$$\mathbf{abort} \mathrel{\widehat{=}} \{\mathit{false}\}$$

The program **abort** is thus undefined for any input instantiations.

Program Operators. The disjunction of two programs $(S \lor T)$ behaves similarly to logical disjunction. The output instantiations of a disjunction is the union of the instantiations of the two programs. There are two forms of conjunction: a parallel version $(S \land T)$, where S and T are executed independently and the intersection of their instantiations is formed on completion; and a sequential form (S, T), where S is executed before T, and hence T can rely on the context established by S.

Quantifiers. The existential quantifier $(\exists\ V \bullet S)$ generalises disjunction, computing the union of the results of S for all possible values of V. Similarly, the universal quantifier $(\forall\ V \bullet S)$ generalises conjunction, computing the intersection of the results of S for all possible values of V.

Procedure call. A procedure call is of the form $pc(U)$, where pc is a procedure and U is a list of terms.

V **:-** S	-	procedure
re p • V **:-** $\mathcal{C}(p)$ **er**	-	recursive procedure
$id \mathrel{\widehat{=}} proc$	-	procedure definition

Fig. 2. Summary of procedure definitions

2.2 Procedure definitions

A summary of the syntax associated with procedures is given in Fig. 2.

Procedures. A (non-recursive) procedure is of the form V **:-** S, where V is a list of formal parameters and S is the body of the procedure (a command).

Recursive procedure. A recursive procedure has the form **re** p • V **:-** $\mathcal{C}(p)$ **er**. Its body, $\mathcal{C}(p)$, encodes zero or more recursive calls to p. To guarantee termination, the actual parameters of the recursive calls must be less than the formal parameters (V) according to some well-founded relation.

Procedure definition. A procedure definition is of the form $id \mathrel{\widehat{=}} proc$, where id is the name of the procedure and $proc$ is a (recursive or non-recursive) procedure.

A distinguishing feature of the refinement calculus when compared to most logic program synthesis schemes is the inclusion of assumptions. This allows one to easily distinguish between what is assumed by a program and what the program must establish. This is useful when defining procedures; often a procedure assumes the type of some of its parameters, e.g., $\{X \in list(\mathbb{N})\}$. This assumption may simplify the refinement — without it some of the desired properties of the parameter cannot be used. Alternatively a procedure may be specified to *establish* the type of one of its parameters, by giving the type in a specification rather than an assumption, e.g., $\langle X \in list(\mathbb{N})\rangle$. In logic programming terms, in the case where a type is given in an assumption the actual parameter to the procedure must be bound to a term of that type. The actual parameter must satisfy whatever assumptions are made about it, or the procedure may abort.

Example. We may specify a procedure *reverse* that relates a list with its reverse. We assume list indices start at 1.

$$
\begin{array}{l}
reverse \mathrel{\widehat{=}} (L, R)\textbf{:-} \\
\quad \{list(L)\}, \\
\quad \langle list(R) \land \#L = \#R\rangle \land \\
\quad \langle(\forall\, i{:}\, 1..\#L \bullet L(i) = R((\#L - i) + 1))\rangle
\end{array}
$$

We have defined *reverse* to be a procedure with formal parameters L and R. Within the body of the definition, we assume that L is a list, giving the type of L as well as ensuring that L must be bound before a call to *reverse*. The procedure is then required to establish that R is a list of the same size as L, and that the elements of R are the same as those of L, but in reverse order.

A more concrete implementation of the *reverse* specification is given by the following recursive program[1].

Definition 2.1 (*Reverse of a list*)

$$\begin{array}{l} \mathit{reverse} \sqsubseteq \mathbf{re}\ \mathit{rev} \bullet (L, R)\textbf{:-} \\ \quad \langle L = [\,] \wedge R = [\,] \rangle \vee \\ \quad (\exists\, H, T \bullet \langle L = [H \mid T] \rangle, \\ \qquad (\exists\, RT \bullet \mathit{append}(RT, [H], R) \wedge \mathit{rev}(T, RT)))\ \mathbf{er} \end{array}$$

We have a recursive block that uses the name *rev* for recursive calls. The body is a disjunction; the first disjunct is the base case where L is empty, and therefore R is also empty. The second disjunct is the recursive case, where L is nonempty. We reverse the tail of L with the recursive call $\mathit{rev}(T, RT)$, and append the head of L, H, onto the end of RT (*append* defines the relationship between three lists where the third is the concatenation of the first two).

2.3 Refinement

Program S is refined by program T, written $S \sqsubseteq T$, if T aborts less often than S, and when S does not abort, T produces the same answers as S. Program equivalence ($\sqsupseteq\sqsubseteq$) is defined as refinement in both directions.

This definition of refinement does not allow the reduction of nondeterminism that imperative refinement allows; in logic programming we are interested in *all* possible solutions, and hence any refinement must also return all of those solutions.

2.4 Refinement laws

In this section we present some basic refinement laws[2]. Each law represents a refinement (synthesis/transformation) that may be made. Where a law is divided into two parts by a horizontal line, the part above the line is the proof obligation that must be satisfied for the refinement below the line to be applied. For example, Law 1 (*weaken assumption*) allows an assumption $\{A\}$ to be refined to $\{B\}$, if A entails B. This corresponds to reducing the conditions under which the program can abort. Law 2 (*equivalent specifications*) allows the characteristic predicate of a specification to be replaced with an equivalent predicate. This corresponds to maintaining the set of answers for free variables. These two laws embody the definition of refinement; they are the main laws we use for manipulating predicates.

Law 1 (*Weaken assumption*)

$$\frac{A \Rrightarrow B}{\{A\} \sqsubseteq \{B\}}$$

Law 2 (*Equivalent specifications*)

$$\frac{P \equiv Q}{\langle P \rangle \sqsupseteq\sqsubseteq \langle Q \rangle}$$

[1] A refinement of the abstract *reverse* definition to the recursive version may be found in Colvin (2002).

[2] All refinement laws used in this paper have been proved correct with respect to the semantics of the language (Colvin 2002).

An entailment $A \Rrightarrow B$ holds if and only if $A \Rightarrow B$ holds for all possible values of the free variables in the predicates A and B. The equivalence operator $\equiv$ is defined as entailment in both directions.

Law 3 (*monotonicity of parallel conjunction*) is an example of a monotonicity law. In general, a monotonicity law states that the refinement of a component of a program refines the entire program. In this case, if S refines to S' and T refines to T' then the parallel conjunction $S \wedge T$ refines to $S' \wedge T'$. Monotonicity holds for all the operators and both quantifiers in the wide-spectrum language.

Law 3 (*Monotonicity of parallel conjunction*)

$$\frac{S \sqsubseteq S';\quad T \sqsubseteq T'}{S \wedge T \sqsubseteq S' \wedge T'}$$

3 Contextual refinement

During refinement we often focus on a component of a program and refine it, resulting in a refinement of the entire program, i.e., our wide-spectrum language is monotonic with respect to refinement. In many situations the larger program can provide context that assists in the refinement of a component. This context can be used, for instance, to discharge proof obligations. In this section we introduce a general notion of context to the calculus, and demonstrate its use with the refinement of a list-reversal procedure. The approach taken is particularly useful when using a refinement tool, as demonstrated in Hemer et al. (2001). The tool can manage the context, instead of the user having to explicitly pass the context around in the form of assumptions.

3.1 Context in refinement laws

Some laws, such as Law 2 (*equivalent specifications*) are "stand-alone" laws. Its premise, $P \equiv Q$, requires that P must be equivalent to Q, regardless of the context in which it appears. However, we may wish to reference the context in order to discharge this proof obligation. To do this, we introduce a generalised form of Law 2.

$$\frac{A \Rrightarrow (P \Leftrightarrow Q)}{\{A\}, \langle P \rangle \sqsupseteq \{A\}, \langle Q \rangle}$$

This law allows assumptions to be used in the proof that P is equivalent to Q. We say the specification $\langle P \rangle$ has A in context. Since we often encounter laws where a refinement occurs with respect to some context we introduce an abbreviation.

$$A \Vdash S \sqsubseteq S' \;\widehat{=}\; \{A\}, S \sqsubseteq \{A\}, S'$$

This is similar to the notation used by Nickson and Hayes (1997) for contextual refinement of imperative programs. Thus the generalised form of Law 2 is written as

Law 4 (*Equivalent specifications w.r.t. context*)

$$\frac{A \Rrightarrow (P \Leftrightarrow Q)}{A \Vdash \langle P \rangle \sqsupseteq \langle Q \rangle}$$

The following law is similar to Law 3 (*monotonicity of parallel conjunction*), except that context for a parallel conjunction is inherited by both conjuncts.

Law 5 (*Monotonicity of parallel conjunction*)

$$\frac{(A \Vdash S \sqsubseteq S');\ \ (A \Vdash T \sqsubseteq T')}{A \Vdash S \wedge T \sqsubseteq S' \wedge T'}$$

To refine $S \wedge T$ in context A, we may refine either of the conjuncts S or T using A as the context. There are similar contextual monotonicity laws for the other constructs in our language. Such laws allow the context to be passed around in a straightforward manner, and for this reason we do not explicitly mention the application of such laws in refinements.

For a sequential conjunction (S, T), command S is executed before T, and hence S establishes a context for T. For example, in the program $\langle X = 1 \rangle, \langle Y = X + 1 \rangle$, the first component establishes $X = 1$, and this may be assumed when refining the second component, e.g., the second component may be refined to $\langle Y = 2 \rangle$. Law 6 gives the general rule when the first component is an assumption, and Law 7 when it is a specification.

Law 6 (*Assumption in context*)

$$\frac{A \wedge B \Vdash (T \sqsubseteq T')}{A \Vdash \{B\}, T \sqsubseteq \{B\}, T'}$$

Law 7 (*Specification in context*)

$$\frac{A \wedge P \Vdash (T \sqsubseteq T')}{A \Vdash \langle P \rangle, T \sqsubseteq \langle P \rangle, T'}$$

Using Law 6 we may refine T with B in context in addition to A, and similarly with P in Law 7. This information may be used to discharge proof obligations in the refinement of T to T'.

3.2 Contextual data refinement

In this section we use contextual refinement to demonstrate *data refinement*, where a variable of an *abstract* type is replaced with one or more variables of a *concrete* type. Data refinement may be used to replace a specification type with an implementation type, or to improve the efficiency of a program. The abstract and concrete types are related by a *coupling invariant*, which is used to provide context for the data refinement. As an example, we show part of the refinement of the simple implementation of *reverse* (Definition 2.1) on lists to a more efficient implementation using difference lists (sometimes referred to as an accumulator implementation). In Sect. 3.2.1 we data refine *reverse* assuming that the couping invariant holds in context; in Sect. 3.2.2 we complete the data refinement by showing how the coupling invariant context can be established efficiently and transparently.

3.2.1 Coupling invariant in context

We refine a procedure call $reverse(L, R)$ in a context in which the list R is represented by the difference list $(DL1, DL2)$, i.e.,

$$R \frown DL2 = DL1 \tag{3.1}$$

The operator '$\frown$' represents list concatenation, thus R is a prefix of $DL1$ and $DL2$ is a suffix of $DL1$. When this relationship holds, $R = DL1 - DL2$ (interpreting '$-$' as list difference).

We begin the refinement of $reverse(L, R)$, with the coupling invariant as an assumption (the context for the refinement).

$$\{R \frown DL2 = DL1\}, reverse(L, R)$$

We expand the call $reverse(L, R)$ from Definition 2.1.

$$\begin{array}{l}\{R \frown DL2 = DL1\},\\ \langle L = [\,] \wedge R = [\,]\rangle \vee\\ (\exists\, H, T \bullet \langle L = [H \mid T]\rangle,\\ \quad (\exists\, RT \bullet append(RT, [H], R) \wedge rev(T, RT)))\end{array}$$

Because program disjunction is monotonic with respect to refinement and the context of the disjunction is inherited by its disjuncts, we may refine the first disjunct, $\langle L = [\,] \wedge R = [\,]\rangle$, with the coupling invariant in context. Using Law 4 (*equivalent specifications*) we rewrite $R = [\,]$ to $DL1 = DL2$, since the context (3.1) implies they are equivalent expressions.

$$\langle L = [\,] \wedge DL1 = DL2\rangle$$

The details of the refinement of the second disjunct are more complex, requiring the introduction of a recursive call. We omit the details for brevity, though the full refinement can be found in Colvin (2002). The resulting recursive program is the usual difference list implementation of *reverse*.

$$\begin{array}{l}reversedl \mathrel{\widehat{=}} \mathbf{re}\, revdl \bullet (L, DL1, DL2)\mathbf{:\text{-}}\\ \quad \langle L = [\,] \wedge DL1 = DL2\rangle \vee\\ \quad (\exists\, H, T \bullet \langle L = [H \mid T]\rangle, revdl(T, DL1, [H \mid DL2]))\,\mathbf{er}\end{array}$$

The refinement can be summarised by the following relation:

$$R \frown DL2 = DL1 \Vdash reverse(L, R) \sqsubseteq reversedl(L, DL1, DL2) \tag{3.2}$$

3.2.2 Data refinement by establishing context

In the previous section a context was given that allows calls to *reverse* to be replaced with calls to the more efficient procedure *reversedl*. However establishing this context in arbitrarily large and complex programs may not be feasible. In this section we show how the problem can be avoided by implementing *reverse* in terms of *reversedl*.

We start by choosing a stronger coupling invariant than (3.1), in which $DL1$ is equal to R and $DL2$ is the empty list.

$$R = DL1 \wedge DL2 = [\,] \tag{3.3}$$

Hence we may deduce $reverse(L, R) \sqsubseteq reversedl(L, R, [\,])$ because (3.3) implies the premise of (3.2). This is a valid refinement in any context. Of course, in a program that makes many calls to *reverse*, we may hide this change by implementing the body of *reverse* as just a call to $reversedl(L, R, [\,])$. The (new) body of *reverse* provides the context of (3.3) locally, avoiding the need for the calling program to establish the context.

The above refinements are examples of *data refinement* on procedures. In the next section we consider data refinement on groups of procedures that operate on a common data type.

4 Modular logic program refinement

In this section we introduce the notion of a *module*, which is a group of procedures that operate on a common data type. By making some assumptions about the context in which an abstract module may be used, we may allow a more efficient module to be used in its place.

4.1 Module specifications

As with modules in logic programming languages such as Mercury (Somogyi et al. 1995) and Gödel (Hill and Lloyd 1994), modules in the wide-spectrum language are collections of procedures that operate on a common data type. The data type is intended to be *opaque*, that is, the implementation of the type is hidden, and variables of that type may only be manipulated via the procedures of the module.

We split the opaque parameters of a module procedure into two categories, *input* and *output*, which correspond with the logic programming *modes* "ground" and "var" (unbound), respectively. Upon a procedure call, opaque inputs must already have been instantiated to the module type and opaque outputs must be uninstantiated. In addition, procedures may have a set of *regular*, i.e., non-opaque, parameters.

Fig. 3 defines a module *Partial Function* that declares operations on a type pfun. The type pfun is a partial function from elements of its domain type σ to elements of its range type τ, written $\sigma \nrightarrow \tau$. A function may be modeled as a set of pairs. A partial function is a function that may be undefined for some elements of its domain, as distinct from a total function which maps every element of its domain to some value. We have left the actual types for σ and τ unspecified since none of the operations depend on these types (though later we will assume that a hash function exists for σ) — we can therefore consider *Partial Function* to be polymorphic. Within the module the type signature of each procedure is declared. Opaque inputs have an assumption about their type and the specification of each procedure guarantees that the opaque outputs are instantiated to be of the opaque

Module *Partial Function*
Type $\mathsf{pfun} \mathrel{\widehat{=}} \sigma \nrightarrow \tau$

$$
\begin{array}{rl}
\mathit{init}: & F' : \mathsf{pfun}_o \\
\mathit{update}: & K : \sigma,\ V : \tau,\ F : \mathsf{pfun}_i,\ F' : \mathsf{pfun}_o \\
\mathit{access}: & K : \sigma,\ F : \mathsf{pfun}_i,\ V : \tau \\
\mathit{remove}: & K : \sigma,\ F : \mathsf{pfun}_i,\ F' : \mathsf{pfun}_o
\end{array}
$$

$$
\begin{array}{rl}
\mathit{init} \mathrel{\widehat{=}} & F' \mathbin{:\!\text{-}} \langle F' = \{\} \rangle \\
\mathit{update} \mathrel{\widehat{=}} & (K, V, F, F') \mathbin{:\!\text{-}} \{F \in \mathsf{pfun} \land K \in \sigma \land V \in \tau\}, \langle F' = F \oplus \{(K, V)\} \rangle \\
\mathit{access} \mathrel{\widehat{=}} & (K, F, V) \mathbin{:\!\text{-}} \{F \in \mathsf{pfun} \land K \in \sigma\}, \langle K \in \mathrm{dom}(F) \land V = F(K) \rangle \\
\mathit{remove} \mathrel{\widehat{=}} & (K, F, F') \mathbin{:\!\text{-}} \{F \in \mathsf{pfun} \land K \in \sigma\}, \langle F' = \{K\} \mathrel{\lhd\!\!\!-} F \rangle
\end{array}
$$

End

Fig. 3. Abstract partial function module

type. Opaque inputs and outputs are subscripted with i and o, respectively. The parameters of type σ and τ (K and V) are regular parameters.

In the definition of *update*, the symbol '$\oplus$' stands for function override; the function $f \oplus g$ is the same as function f, except with all elements in the domain of g mapped according to g. Therefore, $F \oplus \{(K, V)\}$ is the same as F but with K mapped to V instead of $F(K)$. In the definition of *remove* we use domain subtraction '$\mathrel{\lhd\!\!\!-}$'; the function $\{K\} \mathrel{\lhd\!\!\!-} F$ is the same as F, except K is no longer in the domain.

Following the data type terminology of Liskov and Guttag (1986), a procedure with no opaque inputs is referred to as an *initialisation* procedure; for example, *init* is an initialisation procedure which instantiates the opaque output F' to the empty function (represented by the empty set of pairs). A procedure with no opaque outputs is referred to as an *observer*; for example, *access* is an observer that fails if the regular parameter K is not in the domain of the opaque input function F, and instantiates the regular parameter V to $F(K)$ otherwise. A procedure with both opaque inputs and outputs is called a *constructor*; for example, the procedure *update* has an opaque output F', which is the opaque input F updated by the pair (K, V). A constructor can be likened to updating the state in an imperative module. Note that *init*, *update*, and *remove* all guarantee that their opaque output is an element of pfun.

4.2 Using modules

Our intuition is that a module is to be used opaquely in the construction and maintenance of some data structure throughout multiple procedure calls. We therefore consider programs whose procedure calls are ordered so that the intended modes of the opaque inputs and outputs are satisfied, and the variables used as opaque inputs and outputs are local to the program. For instance, consider the following program that uses the module *Partial Function*. It inserts the pairs $(a, 2)$ and $(b, 1)$

into a function and accesses the value for a.

$$\begin{array}{l}(\exists\, F \bullet \mathit{init}(F), (\exists\, F' \bullet \mathit{update}(a, 2, F, F'),\\ \qquad (\exists\, F'' \bullet \mathit{update}(b, 1, F', F''), \mathit{access}(a, F'', X))))\end{array} \tag{4.1}$$

The use of sequential conjunction reflects the notion of the changing state and also allows the assumptions of the later calls to be satisfied. Initially, F is instantiated to the empty function. The two calls to *update* update F to F' and then to F''. Overall, the only variable we are interested in is X — the opaque parameters are local because they are existentially quantified when they are used as an output. By only dealing with programs of this form, we can use contextual information to derive more efficient implementations of the module.

To formalise this notion, we say a program is in *output-quantified form* with respect to a module $\mathcal{M}$ if, for all procedure calls $p(V, I, O)$ where p is in $\mathcal{M}$ and V stands for the regular parameters, the opaque inputs I are bound and the opaque outputs O are not bound before the call. Also, the opaque variables must not be used except by procedures in $\mathcal{M}$. We first define open output-quantified form, which is a generalisation of output-quantified form.

Definition 4.2 (*Open output-quantified form*)
We say a program is in *open output-quantified form* w.r.t. a module $\mathcal{M}$ and a set of free opaque variables IV if it is in one of the following forms:

1. a program fragment that does not rely on the opaque variables in IV nor make calls on any of the procedures in $\mathcal{M}$;
2. a program of one of the following forms,

   $\mathcal{C}_1 \vee \mathcal{C}_2$
   $\mathcal{C}_1 \wedge \mathcal{C}_2$
   $\mathcal{C}_1, \mathcal{C}_2$
   $(\exists\, V \bullet \mathcal{C}_1)$
   $(\forall\, V \bullet \mathcal{C}_1)$

   where $\mathcal{C}_1$ and $\mathcal{C}_2$ are subcomponents that are in open output-quantified form w.r.t. $\mathcal{M}$ and IV, and V is a regular (non-opaque) variable; or,
3. a program of the form

   $$(\exists\, O \bullet p(V, I, O), \mathcal{C})$$

   where p is a procedure in $\mathcal{M}$ and V, I, and O are the regular, opaque input, and opaque output parameters, respectively, of p. The opaque inputs I must be a subset of IV. The component $\mathcal{C}$ must be in open output-quantified form w.r.t. $\mathcal{M}$ and the set $IV \cup \{O\}$. When p has no outputs, i.e., it is an observer, there are no quantified variables and the corresponding form is just $p(V, I), \mathcal{C}$.

Definition 4.3 (*Output-quantified form*)
We say a program is in *output-quantified form* w.r.t. a module $\mathcal{M}$ if it is in open output-quantified form w.r.t. $\mathcal{M}$ and contains no free opaque variables.

Because logic programs do not typically have "state", we must pass the opaque parameters explicitly, and hence in some sense the implementation details are exposed. However, programs that are in output-quantified form are restricted to only using the opaque type and variables via the procedures of the module. This ensures the module is used as intended, i.e., with the type being opaque.

Since the type is opaque, a program in output-quantified form is amenable to syntactic simplification that hides the opaque variables. The opaque variables are locally quantified, and typically appear as input/output pairs, thus we can adopt a shorthand similar to that of definite clause grammars (DCGs) in Prolog (and other logic programming languages). For instance, we could write program (4.1) thus:

$$\mathit{init}, \mathit{update}(a, 2), \mathit{update}(b, 1), \mathit{access}(a, X) \tag{4.4}$$

At each call to a procedure from the module, from form 3 in Definition 4.2 we can immediately identify that a new output opaque variable must be quantified (except in the case of the observer, *access*), and fill in the in/output parameters of the procedure call appropriately (resulting in program (4.1)). However, syntactic simplifications like this restrict expressiveness. For instance, by hiding the state we have no easy way of having two instances of the state active at one time (imperative languages without opaque types also have this problem). For instance, the shorthand notation cannot be used to simplify the following program, which has two different partial functions G and H, containing $(b, 1)$ and $(b, 2)$, respectively.

$$(\exists\, F \bullet \mathit{init}(F), (\exists\, G \bullet \mathit{update}(b, 1, F, G), (\exists\, H \bullet \mathit{update}(b, 2, F, H), \ldots))$$

For this reason we use the more general notation in which opaque variables are explicit.

4.3 Module refinement

In general, we say a module $\mathcal{M}$ is refined by a module $\mathcal{M}^+$ if, for all possible programs S using calls to $\mathcal{M}$, S is refined by the program S^+ obtained by replacing all calls to the procedures of $\mathcal{M}$ by calls to the corresponding procedures of $\mathcal{M}^+$. In this section we consider a law for module refinement (Theorem 4.5) that can be used only if the programs using the module are in output-quantified form (Definition 4.3).

Consider the *Partial Function* module defined in Fig. 3. A program that uses it, e.g., (4.1), is not directly implementable, since the module uses the abstract partial function type which is not part of the implementation language. We would like to replace the calls to *init*, *update*, *remove*, and *access* from the *Partial Function* module with corresponding calls on a module that implements the operations on an implementation data type. Of course, replacing the references to the *Partial Function* module with references to the implementation module must result in a refinement of the program in question. The following is our theorem for module refinement. As with the data refinement example in Sect. 3.2, we require a coupling invariant (CI) to relate the abstract and concrete types.

Theorem 4.5 (*Module Refinement*)

Assume the following: modules $\mathcal{M}$ and $\mathcal{M}^+$, with associated opaque types Σ and Σ^+, respectively; a coupling invariant CI, that relates the types Σ and Σ^+; and all corresponding pairs of procedures p and p^+ from $\mathcal{M}$ and $\mathcal{M}^+$, respectively, satisfy Condition 4.6, below, using CI. Then a program, $\mathcal{C}$, which is in output-quantified form w.r.t. $\mathcal{M}$, is refined by the program $\mathcal{C}^+$, which is structurally the same as $\mathcal{C}$ except with procedure calls to module $\mathcal{M}$ replaced by corresponding procedures calls to module $\mathcal{M}^+$.

Proof. The theorem is proved by structural induction over programs in open output-quantified form. A detailed proof can be found in (Colvin 2002); it is a generalised version of the proof in (Colvin et al. 2001). □

Consider the abstract and concrete procedures p and p^+ which are defined as follows.

$$p \mathrel{\widehat{=}} (V, I, O) \textbf{ :- } \{A\}, \langle P \rangle$$
$$p^+ \mathrel{\widehat{=}} (V, I^+, O^+) \textbf{ :- } \{A^+\}, \langle P^+ \rangle$$

The variables in I and O are of the abstract opaque type Σ, and similarly the variables in I^+ and O^+ are of the concrete opaque type Σ^+. The regular variables, V, may be of any other type. The free variables of the assumption $\{A\}$ are restricted to V and I, and the free variables of the specification $\langle P \rangle$ are restricted to V, I and O. Corresponding restrictions apply to $\{A^+\}$ and $\langle P^+ \rangle$. The following predicate describes the conditions that must hold between procedures p and p^+ with respect to the coupling invariant CI.

Condition 4.6

$$CI(I,I^+) \wedge A \Rrightarrow \tag{4.7}$$
$$A^+ \wedge \tag{4.8}$$
$$(P \Rightarrow (\exists\, O^+ \bullet P^+ \wedge CI(O, O^+))) \wedge \tag{4.9}$$
$$(P^+ \Rightarrow (\exists\, O \bullet P \wedge CI(O, O^+))) \tag{4.10}$$

This condition states that, assuming the inputs are related by the coupling invariant and the abstract assumption A holds (4.7): the concrete assumption holds (4.8); every abstract answer has a corresponding answer in the concrete implementation (4.9); and every concrete answer is related to an answer in the abstract procedure (4.10).

4.4 Example

Concrete type. For our implementation of the *Partial Function* module, we assume the existence of an injection[3], *hash*, that uniquely maps elements of type σ to a natural number in the range $0..N-1$. With this assumption, we may implement

[3] By requiring *hash* to be an injection we are assuming that no two keys will map to the same natural number and hence avoid the problem of clashes. A more general approach that handles clashes is possible, but would complicate the presentation.

a partial function as an array, the indices of which are the hashed values of σ. In other words, the array acts as a hash table. We define the type $\mathsf{hashtable}$ as an array of size N, the elements of which are either the range type τ or the special element null (not an element of τ).

$$\mathsf{hashtable} \mathrel{\widehat{=}} (0..N-1) \to (\tau \cup \{\mathsf{null}\})$$

The symbol '$\to$' indicates a total function, which in this case models an array.

Coupling invariant. Now that we have defined the concrete type, we give a coupling invariant that relates a partial function F to a hash table H:

$$H = \mathit{makehash}(F) \tag{4.11}$$

where $\mathit{makehash}(F) = \{i{:}\, 0..N-1 \bullet (i, \mathsf{null})\} \oplus \{(K, V){:}\, F \bullet (\mathit{hash}(K), V)\}$. We have written *makehash* as a *set comprehension*. In general, a set comprehension $\{x{:}\, T \bullet e(x)\}$ represents the set of values of the expression $e(x)$ for each element x of type T. For example, $\{i{:}\, 0..N-1 \bullet (i, \mathsf{null})\}$ is the set of pairs (i, null) for each number i in the range $0..N-1$. Thus $\mathit{makehash}(F)$ is a mapping from $\mathit{hash}(K)$ to V for all pairs (K, V) appearing in the function F, with all other numbers mapping to null. We assume we have available a module that implements operations such as updates and accesses on arrays in constant time, e.g., the `array` module in Mercury (Somogyi et al. 1995). We note the following property:

$$F \in \mathsf{pfun} \wedge H = \mathit{makehash}(F) \Rightarrow H \in \mathsf{hashtable} \tag{4.12}$$

Condition 4.6 for procedure update. As an example instantiation of Condition 4.6, we prove that, given the coupling invariant (4.11), the following procedure is a valid array implementation of *update* (from Fig. 3).

$$\begin{aligned} &\mathit{update} \mathrel{\widehat{=}} (K, V, H, H')\textbf{:-} \\ &\quad \{H \in \mathsf{hashtable} \wedge K \in \sigma \wedge V \in \tau\}, \langle H' = H \oplus \{(\mathit{hash}(K), V)\}\rangle \end{aligned}$$

This can be implemented efficiently in Mercury by using the `set` predicate from the `array` module.

First we show (4.7) entails (4.8).

$$\begin{aligned} &H = \mathit{makehash}(F) \wedge F \in \mathsf{pfun} \wedge K \in \sigma \wedge V \in \tau \Rrightarrow \\ &\quad H \in \mathsf{hashtable} \wedge K \in \sigma \wedge V \in \tau \end{aligned}$$

The conditions on K and V hold trivially, and $H \in \mathsf{hashtable}$ follows from (4.12). We would normally expect (4.8) to be shown this easily.

Now we show the rest of Condition 4.6 holds.

$$\begin{aligned} &H = \mathit{makehash}(F) \wedge F \in \mathsf{pfun} \wedge K \in \sigma \wedge V \in \tau \Rrightarrow \\ &\quad (F' = F \oplus \{(K, V)\} \Rightarrow \\ &\qquad (\exists\, H' \bullet H' = H \oplus \{(\mathit{hash}(K), V)\} \wedge H' = \mathit{makehash}(F'))) \wedge \\ &\quad (H' = H \oplus \{(\mathit{hash}(K), V)\} \Rightarrow \\ &\qquad (\exists\, F' \bullet F' = F \oplus \{(K, V)\} \wedge H' = \mathit{makehash}(F'))) \end{aligned}$$

Simplifying using the one-point rule we get:

$$\begin{array}{l} H = \mathit{makehash}(F) \land F \in \mathsf{pfun} \land K \in \sigma \land V \in \tau \Rrightarrow \\ \quad (F' = F \oplus \{(K, V)\} \Rightarrow H \oplus \{(\mathit{hash}(K), V)\} = \mathit{makehash}(F')) \land \\ \quad (H' = H \oplus \{(\mathit{hash}(K), V)\} \Rightarrow H' = \mathit{makehash}(F \oplus \{(K, V)\})) \end{array}$$

Now we simplify the implications, combining them into a single stronger predicate.

$$\begin{array}{l} H = \mathit{makehash}(F) \land F \in \mathsf{pfun} \land K \in \sigma \land V \in \tau \Rrightarrow \\ \quad H \oplus \{(\mathit{hash}(K), V)\} = \mathit{makehash}(F \oplus \{(K, V)\}) \end{array}$$

Thus we must show that, given that the coupling invariant for the inputs F and H holds and that F and the regular parameters are of the correct type, the coupling invariant holds for the output values. We prove the conclusion by manipulating the expression $H \oplus \{(\mathit{hash}(K), V)\}$.

$$\begin{array}{ll} & H \oplus \{(\mathit{hash}(K), V)\} \\ = & \text{from antecedent } H = \mathit{makehash}(F); \text{ definition of } \mathit{makehash} \\ & \{i : 0..N-1 \bullet (i, \mathsf{null})\} \oplus \{(X, Y) : F \bullet (\mathit{hash}(X), Y)\} \oplus \{(\mathit{hash}(K), V)\} \\ = & \text{Since } F \text{ is a function and } \mathit{hash} \text{ is an injection} \\ & \{i : 0..N-1 \bullet (i, \mathsf{null})\} \oplus \{(X, Y) : F \oplus \{(K, V)\} \bullet (\mathit{hash}(X), Y)\} \\ = & \text{Definition} \\ & \mathit{makehash}(F \oplus \{(K, V)\}) \end{array}$$

Thus we have proved Condition 4.6 for *update*. The full array implementation of the *Partial Function* module is shown later (Fig. 5); the remaining procedures are derived using techniques described in Sect. 5.

5 Calculating a concrete module

In the previous section we described the conditions that must hold between two modules with respect to a coupling invariant to allow module refinement. In this section we show how those conditions may be used to calculate a concrete module, given an abstract module and an appropriate coupling invariant. The procedures of the calculated module are guaranteed to satisfy Condition 4.6 with respect to their corresponding abstract procedures. After introducing the general form of a calculated concrete procedure, we specialise the technique based on the determinism of the coupling invariant and the abstract procedure.

5.1 General form of concrete procedure

The following theorem gives the general form for the concrete procedure given the abstract procedure and the coupling invariant.

Theorem 5.1 (*Module calculation*)
Given a procedure $p \mathrel{\widehat{=}} (V, I, O) \mathbf{:-} \{A\}, \langle P \rangle$ with at most V and I free in A, and at

most V, I and O free in P, if p and the coupling invariant CI satisfy the following properties for some predicate R which is independent of I,

$$CI(I, I^+) \wedge A \wedge P \Rrightarrow (\exists\, O^+ \bullet CI(O, O^+)) \tag{5.2}$$

$$CI(I, I^+) \wedge A \Rrightarrow (\exists\, O \bullet P \wedge CI(O, O^+)) \Leftrightarrow R(V, I^+, O^+) \tag{5.3}$$

then the following implementation of the concrete procedure satisfies Condition 4.6.

$$\begin{array}{l} p^+ \mathrel{\widehat{=}} (V, I^+, O^+) \textbf{:-} \\ \quad \{(\exists\, I \bullet CI(I, I^+) \wedge A)\}, \\ \quad \langle (\forall\, I \bullet CI(I, I^+) \wedge A \Rightarrow (\exists\, O \bullet P \wedge CI(O, O^+)))\rangle \end{array} \tag{5.4}$$

With this theorem we may immediately derive a concrete module from an abstract module that will satisfy Condition 4.6, provided the coupling invariant satisfies (5.2) and (5.3).

Proof. [4] To prove that (5.4) satisfies Condition 4.6, we prove that it satisfies (4.8), (4.9) and (4.10), assuming (5.2), (5.3), (4.7), and that the outputs O and O^+ do not occur free in the assumptions A and A^+, respectively.

(4.8) Substitute $(\exists\, I \bullet CI(I, I^+) \wedge A)$ for A^+ in (4.8); then (4.8) holds from (4.7).

(4.10) Substituting $(\forall\, I \bullet CI(I, I^+) \wedge A \Rightarrow (\exists\, O \bullet P \wedge CI(O, O^+)))$ for P^+ in (4.10), with bound variable I renamed to X, gives the following.

$$\begin{array}{l} (\forall\, X \bullet CI(X, I^+) \wedge A[\frac{X}{I}] \Rightarrow (\exists\, O \bullet P[\frac{X}{I}] \wedge CI(O, O^+))) \Rightarrow \\ \quad (\exists\, O \bullet P \wedge CI(O, O^+)) \end{array}$$

Since we have $CI(I, I^+) \wedge A$ in context (4.7), from the implication of the universally quantified predicate we may deduce $(\exists\, O \bullet P \wedge CI(O, O^+))$.

(4.9) Substituting $(\forall\, I \bullet CI(I, I^+) \wedge A \Rightarrow (\exists\, O \bullet P \wedge CI(O, O^+)))$ for P^+ in (4.9), with variable renaming to avoid clashes, gives the following.

$$\begin{array}{l} P \Rightarrow (\exists\, O^+ \bullet \\ \quad (\forall\, X \bullet CI(X, I^+) \wedge A[\frac{X}{I}] \Rightarrow (\exists\, Y \bullet P[\frac{X,Y}{I,O}] \wedge CI(Y, O^+))) \wedge \\ \quad CI(O, O^+)) \end{array}$$

We simplify the middle line to *true*, assuming P and $CI(O, O^+)$.

$$\begin{array}{ll} & (\forall\, X \bullet CI(X, I^+) \wedge A[\frac{X}{I}] \Rightarrow (\exists\, Y \bullet P[\frac{X,Y}{I,O}] \wedge CI(Y, O^+))) \\ \Leftrightarrow & \text{We use (5.3) on the quantification over } Y. \\ & (\forall\, X \bullet CI(X, I^+) \wedge A[\frac{X}{I}] \Rightarrow R(V, I^+, O^+)) \\ \Leftrightarrow & \text{Reduce the scope of } X \\ & (\exists\, X \bullet CI(X, I^+) \wedge A[\frac{X}{I}]) \Rightarrow R(V, I^+, O^+) \\ \Leftrightarrow & X \text{ is witnessed by } I \text{ from (4.7)} \\ & R(V, I^+, O^+) \\ \Leftrightarrow & \text{We now use (5.3) again, from (4.7) in context} \end{array}$$

[4] This is a simplified version of the proof that originally appeared in Colvin (2002).

$$
\begin{array}{ll}
 & (\exists\, Y \bullet P[\frac{Y}{O}] \wedge CI(Y, O^+)) \\
\Leftrightarrow & Y \text{ is witnessed by } O \text{ from assumptions } P \text{ and } CI(O, O^+) \\
 & \mathit{true}
\end{array}
$$

With the middle line simplified, we are left with

$$P \Rightarrow (\exists\, O^+ \bullet CI(O, O^+))$$

Using (4.7) from the context, this follows from (5.2).

□

The first assumption (5.2) of Theorem 5.1 requires that the effect of the abstract procedure implies that its output, O, has some concrete representation. This is typically just a type check on O, since it is the only free variable in $(\exists\, O^+ \bullet CI(O, O^+))$. One would always expect (5.2) to hold, and in general it can be trivially discharged. The second assumption (5.3) requires that the expression $(\exists\, O \bullet P \wedge CI(O, O^+))$, in a context including $CI(I, I^+) \wedge A$, has some equivalent form R that does not include a free occurrence of the abstract input I. In practice, one does not need to explicitly discharge (5.3). The given form of the concrete procedure (5.4) still involves the abstract type via the coupling invariant (on both the input and output). One will need to simplify the concrete procedure to remove the abstract data type; once the abstract input has been removed (if possible), (5.3) has been satisfied.

Both constraints (5.2) and (5.3) can be used as a consistency check for the entire abstract module and chosen coupling invariant, prior to calculating the concrete module. As mentioned, condition (5.2) fails for a procedure p if p does not maintain the abstract type for its output as expected by the coupling invariant. Condition (5.3) fails when information in the abstract type is lost in the transformation to the concrete type, and the abstract procedures make use of that information. For instance, consider refining an "abstract" list module to a "concrete" set module, where the coupling invariant is just that the set holds all the elements in the list (thus losing information about how many times an element appears in the list, and the order of elements in the list). We can implement procedures for adding elements and checking membership easily, however we would not expect to be able to implement a *count* procedure, which returns the number of times an element appears in the list. Accordingly, we will not be able to prove (5.3) for the *count* procedure with the chosen coupling invariant and concrete type.

In general, a carefully chosen coupling invariant will ensure (5.2) and (5.3) hold. In particular, a coupling invariant in which the abstract value is some function of the concrete, i.e., $I = af(I^+)$, will always ensure that (5.3) holds. This is because all occurrences of the abstract input can be replaced with $af(I^+)$.

5.1.1 Simplifying the specification

In practice, the calculated specification of a concrete procedure will be simpler than the general form given in (5.4). From (5.3) we know that the right-hand side of the implication can be expressed as $R(V, I^+, O^+)$, which does not contain I or O free. Making this simplification, and reducing the scope of I gives the following simpler

specification part for p^+ in (5.4).

$$\langle (\exists\, I \bullet CI(I, I^+) \wedge A) \Rightarrow R(V, I^+, O^+) \rangle$$

The left-hand side of the implication matches the assumption from (5.4). Using the assumption and Law 4 (*equivalent specifications*) the specification may be simplified to just

$$\langle R(V, I^+, O^+) \rangle$$

Thus, in practice, once the specification has been calculated, it is just a matter of simplifying $(\exists\, O \bullet P \wedge CI(O, O^+))$ to eliminate references to I. Then the universal quantification over I becomes redundant.

5.1.2 Initialisation and observer

The following are instances of (5.4) simplified for initialisations (no opaque inputs) and observers (no opaque outputs), respectively.

$$\{A\}, \langle (\exists\, O \bullet P \wedge CI(O, O^+)) \rangle \tag{5.5}$$

$$\{(\exists\, I \bullet CI(I, I^+) \wedge A)\}, \langle (\forall\, I \bullet CI(I, I^+) \wedge A \Rightarrow P) \rangle \tag{5.6}$$

A consequence of there being no inputs for initialisations is that (5.3) can be trivially satisfied by choosing R to be $(\exists\, O \bullet P \wedge CI(O, O^+))$. Since there are no outputs for an observer there is no need to check (5.2).

5.2 Example

In Sect. 4.4 we provided a proof that a concrete implementation of *update* from Fig. 3 satisfies Condition 4.6. Here we use Theorem 5.1 to calculate an implementation from the abstract *update* procedure and coupling invariant (4.11). We assume that (5.2) and (5.3) hold (an example of discharging these formally will be shown later in Sect. 5.4.1). The concrete procedure in the form of (5.4) is thus:

$$\begin{array}{l} \{(\exists\, F \bullet H = \mathit{makehash}(F) \wedge F \in \mathsf{pfun} \wedge K \in \sigma \wedge V \in \tau)\}, \\ \langle (\forall\, F \bullet H = \mathit{makehash}(F) \wedge F \in \mathsf{pfun} \wedge K \in \sigma \wedge V \in \tau \Rightarrow \\ \quad (\exists\, F' \bullet H' = \mathit{makehash}(F') \wedge F' = F \oplus \{(K, V)\})) \rangle \end{array}$$

From (4.12) we may use Law 1 (*weaken assumption*) to refine the calculated assumption.

$$\{H \in \mathsf{hashtable} \wedge K \in \sigma \wedge V \in \tau\}$$

We simplify the specification by applying the one-point law to F'.

$$\begin{array}{l} \langle (\forall\, F \bullet H = \mathit{makehash}(F) \wedge F \in \mathsf{pfun} \wedge K \in \sigma \wedge V \in \tau \Rightarrow \\ \quad H' = \mathit{makehash}(F \oplus \{(K, V)\})) \rangle \end{array}$$

In Sect. 4.4 we showed

$$\mathit{makehash}(F \oplus \{(K, V)\}) = H \oplus \{(\mathit{hash}(K), V)\}$$

Therefore we may rewrite the bottom line of the specification as

$$H' = H \oplus \{(hash(K), V)\}$$

We have eliminated references to the abstract input F on the right-hand side of the implication. We therefore employ the simplification mentioned in Sect. 5.1.1 to eliminate the quantification over F, resulting in the following program (identical to that given in Sect. 4.4).

$$\{H \in \mathsf{hashtable} \land K \in \sigma \land V \in \tau\}, \langle H' = H \oplus \{(hash(K), V)\}\rangle$$

5.3 Specialisations

In this section we provide some specialisations of (5.4), based on the form of the coupling invariant and the abstract procedure. The specialisations are partitioned based on two factors. Firstly, whether or not the abstract procedure $\{A\}, \langle P\rangle$ is deterministic. In a deterministic procedure there is only one possible abstract output value given any regular and input parameter values, i.e., P is of the form $O = f(V, I)$. In a nondeterministic procedure, there could be many possible output values related to any given regular regular and input values (which we therefore write as a ternary relation $P(V, I, O)$).

Secondly, we partition the specialisations based on the form of the coupling invariant. We consider the case where the abstract variable is some (abstraction) function of the concrete, $I = af(I^+)$. In this situation, there are potentially many concrete representations of an abstract value, though each concrete value represents exactly one abstract value. This is a common form of coupling invariant, and often simplifies the data refinement process. The second form of coupling invariant we consider is when the concrete variable is some (concretisation) function of the abstract variable, $I^+ = cf(I)$. Thus each concrete value may represent many abstract values, though each abstract value has exactly one concrete representation. Finally we consider the case where the coupling invariant is a relation between the abstract and concrete variables, $CI(I, I^+)$.

The specialisations are summarised in Fig. 4. The predicates in the cells of the table are obtained from (5.4) by simplifying using the one-point rule. Most of the transformations are straightforward, however the case where the abstract procedure is deterministic and the coupling invariant involves an abstraction function is discussed in more detail in Sect. 6.

5.4 Example: hash table

In this section we use the derivation process to derive an array (hashtable) implementation of the abstract partial function type given in Fig. 3. Recall the coupling invariant:

$$H = \{i : 0..N-1 \bullet (i, \mathsf{null})\} \oplus \{(K, V) : F \bullet (hash(K), V)\} \tag{4.11}$$

This coupling invariant is a concretisation function (the concrete variable H is a function of the abstract variable F). There is an equivalent *abstraction* function

	Abstract procedure	
Coupling invariant	Deterministic $O = f(V, I)$	Non-deterministic $P(V, I, O)$
$I = af(I^+)$	$af(O^+) = f(V, af(I^+))$	$P(V, af(I^+), af(O^+))$
$I^+ = cf(I)$	$(\forall\, I \bullet I^+ = cf(I) \wedge A \Rightarrow O^+ = cf(f(V, I)))$	no simplification
$CI(I, I^+)$	$(\forall\, I \bullet CI(I, I^+) \wedge A \Rightarrow CI(f(V, I), O^+))$	no simplification

Fig. 4. Specialisations for concrete specification P^+

form (see (5.7) below), but we prefer to use (4.11) for the simplifications it provides in the calculation process. When the abstract procedure is deterministic we may use the simplification from the second row of Fig. 4. The calculated procedures can all be implemented efficiently in the logic programming language Mercury.

5.4.1 Side conditions

Before beginning the derivation, we check that the conditions (5.2) and (5.3) hold for each procedure in the module. Condition (5.2) requires that the coupling invariant on the inputs, as well as A and P of the abstract procedures, imply $(\exists\, O^+ \bullet CI(O, O^+))$. Instantiating the quantification for the hash table example gives the following condition which trivially holds:

$$(\exists\, H \bullet H = \{i{:}\, 0..N-1 \bullet (i, \mathsf{null})\} \oplus \{(K,\, V){:}\, F \bullet (hash(K),\, V)\})$$

It is easily seen that each abstract procedure guarantees that the type of its output parameter is of type pfun, and therefore (5.2) holds for all the procedures in the module.

To satisfy (5.3) we must be able to eliminate references to the abstract type. As mentioned earlier, this side condition is normally satisfied implicitly in the derivation process, since in any case we wish to eliminate the abstract variable. However, we note that in this case there is an equivalent coupling invariant that we could employ:

$$F = (\lambda\, K{:}\, \sigma \mid H(hash(K)) \neq \mathsf{null} \bullet H(hash(K))) \tag{5.7}$$

This coupling invariant expresses the abstract variable F as a function of the concrete variable H. The function is constructed by taking all keys K of type σ which are not mapped to null by the hash table H (the notation '$|$' is used to restrict the domain of a function); all such keys are then mapped to their (non-null) value in the hash table. Because the relationship between the abstract and concrete variables is one-to-one, references to the abstract input can always be eliminated by replacing them with the right-hand side of the equality in (5.7). We may therefore automatically discharge (5.3) for each procedure in the module.

5.4.2 Assumptions

In the partial function module, the assumptions of the procedures are that the input and regular parameters are of the correct type. From (5.4) we calculate the concrete assumption for the *update* procedure.

$$(\exists F \bullet H = makehash(F) \land F \in \mathsf{pfun} \land K \in \sigma \land V \in \tau)$$

From (4.12) we may use Law 1 (*weaken assumption*) to refine the calculated assumption.

$$H \in \mathsf{hashtable} \land K \in \sigma \land V \in \tau$$

Using similar manipulation, the assumption of each concrete procedure is refined to the corresponding abstract assumption, except with $H \in \mathsf{hashtable}$ in place $F \in \mathsf{pfun}$. We now calculate the specification of each concrete procedure, and refine the specification to code (with the exception of $update^+$ which was dealt with in Sect. 5.2).

5.4.3 Procedure init

Since the *init* procedure is a deterministic initialisation and the coupling invariant (4.11) is also deterministic, we may immediately use the simplification in the second row of Fig. 4 with $f(V, I) = \{\}$. Furthermore there are no inputs, eliminating the quantification over I.

$$H' = \{i : 0..N-1 \bullet (i, \mathsf{null})\} \oplus \{(K, V) : \{\} \bullet (hash(K), V)\}$$

The rightmost set comprehension is just the empty set, and therefore the function override has no effect.

$$H' = \{i : 0..N-1 \bullet (i, \mathsf{null})\}$$

In other words, every element in the array is initialised to null.

5.4.4 Procedure remove

This is a deterministic procedure, and we use the specialisation in the second row of Fig. 4. In this case $f(V, I)$ is $\{V\} \lhd I$.

$$\begin{array}{l}(\forall F \bullet H = makehash(F) \land F \in \mathsf{pfun} \land K \in \sigma \Rightarrow \\ \quad H' = \{i : 0..N-1 \bullet (i, \mathsf{null})\} \oplus \{(X, Y) : \{K\} \lhd F \bullet (hash(X), Y)\})\end{array}$$

We simplify the equality on the bottom line. Since F is a function and *hash* is an injection, it is equivalent to

$$H' = \{i : 0..N-1 \bullet (i, \mathsf{null})\} \oplus (\{hash(K)\} \lhd \{(X, Y) : F \bullet (hash(X), Y)\}$$

Therefore $hash(K)$ must map to null in H'.

$$\begin{array}{l}H' = (\{i : 0..N-1 \bullet (i, \mathsf{null})\} \oplus \\ \quad \{(X, Y) : F \bullet (hash(X), Y)\}) \oplus \{(hash(K), \mathsf{null})\}\end{array}$$

We rewrite using the *makehash* function, and make the antecedent explicit again.

$$(\forall F \bullet H = \mathit{makehash}(F) \land F \in \mathsf{pfun} \land K \in \sigma \Rightarrow \\ H' = \mathit{makehash}(F) \oplus \{(\mathit{hash}(K), \mathsf{null})\})$$

From the antecedent we replace $\mathit{makehash}(F)$ with H, eliminating the reference to the abstract input F on the right-hand side of the implication. We therefore use the simplification in Sect. 5.1.1 to eliminate the quantification over F and complete the refinement.

$$H' = H \oplus \{(\mathit{hash}(K), \mathsf{null})\}$$

5.4.5 Procedure access

Since *access* is an observer we instantiate (5.6).

$$(\forall F \bullet H = \mathit{makehash}(F) \land F \in \mathsf{pfun} \land K \in \sigma \Rightarrow \\ K \in \mathrm{dom}(F) \land V = F(K))$$

We manipulate the bottom line.

$$K \in \mathrm{dom}(F) \land V = F(K)$$

Given the assumptions and $K \in \mathrm{dom}(F)$, $F(K) = H(\mathit{hash}(K))$.

$$K \in \mathrm{dom}(F) \land V = H(\mathit{hash}(K))$$

We have that $K \in \mathrm{dom}(F)$ is equivalent to $H(\mathit{hash}(K)) \neq \mathsf{null}$ using (5.7).

$$H(\mathit{hash}(K)) \neq \mathsf{null} \land V = H(\mathit{hash}(K))$$

We simplify.

$$V \neq \mathsf{null} \land V = H(\mathit{hash}(K))$$

As with *remove*, we have eliminated the abstract input from the conclusion of the implication, and therefore eliminate the quantification over F as in Sect. 5.1.1. As expected, the procedure fails rather than return null for V when K is not in the domain. The full module is given in Fig. 5.

6 Non-determinism in module derivations

When dealing with refinement of opaque types (in which the representation of the opaque type is not directly visible), there may be multiple concrete representations of an opaque type variable which are equivalent in terms of the abstract specification. Thus if we choose any one of those representations, the behaviour of the operations on that representation will meet the requirements of the abstract specification. Hence for an opaque variable only one representation from a set of equivalent representations needs to be chosen. This corresponds to *don't care* or *demonic* nondeterminism. At the same time, the abstract specification of an operation may involve *don't know* nondeterminism, where multiple answers, provided

Module *Hashtable*

Type $\mathsf{hashtable} \mathrel{\widehat{=}} 0..N-1 \rightarrow (\tau \cup \{\mathsf{null}\})$

$$
\begin{array}{rl}
\mathit{init}: & H : \mathsf{hashtable}_o \\
\mathit{add}: & K : \sigma,\ V : \tau,\ H : \mathsf{hashtable}_i,\ H' : \mathsf{hashtable}_o \\
\mathit{access}: & K : \sigma,\ H : \mathsf{hashtable}_i,\ V : \tau \\
\mathit{remove}: & K : \sigma,\ H : \mathsf{hashtable}_i,\ H' : \mathsf{hashtable}_o
\end{array}
$$

$$
\begin{array}{rll}
\mathit{init} \mathrel{\widehat{=}} & H \textbf{ :- } \langle H = \{i : 0..N-1 \bullet (i, \mathsf{null})\}\rangle & \\
\mathit{update} \mathrel{\widehat{=}} & (K, V, H, H') \textbf{ :- } & \{H \in \mathsf{hashtable} \land K \in \sigma \land V \in \tau\}, \\
 & & \langle H' = H \oplus \{(\mathit{hash}(K), V)\}\rangle \\
\mathit{access} \mathrel{\widehat{=}} & (K, H, V) \textbf{ :- } & \{H \in \mathsf{hashtable} \land K \in \sigma\}, \\
 & & \langle V \neq \mathsf{null} \land V = H(\mathit{hash}(K))\rangle \\
\mathit{remove} \mathrel{\widehat{=}} & (K, H, H') \textbf{ :- } & \{H \in \mathsf{hashtable} \land K \in \sigma\}, \\
 & & \langle H' = H \oplus \{(\mathit{hash}(K), \mathsf{null})\}\rangle
\end{array}
$$

End

Assume the constants *hash* and N such that *hash* uniquely maps elements of type σ to a natural number in the range $0..N-1$.

Fig. 5. Concrete partial function module

via regular (non-opaque) variables, are possible. Hence in order to handle the information hiding aspects of opaque variables within the logic programming context we need a framework that handles both *don't know* and *don't care* nondeterminism; not just *don't know* (as in standard logic programming) and not just *don't care* (as in concurrent logic programming (Shapiro 1989)).

In this section we apply the basic principles of demonic nondeterminism to module calculation. We apply them to a particular combination of abstract procedure and coupling invariant, for which the calculation method presented in Sect. 5 leads to a procedure that may produce many different answers for the concrete output parameters, though we "don't care" which one is chosen. This reduction in nondeterminism (in the choice of concrete value) will typically lead to a more efficient concrete module.

6.1 Deterministic abstract procedure and an abstraction function

Consider the specialisation in the top-left entry in Fig. 4, where we have a deterministic abstract procedure and an abstraction function as the coupling invariant. The calculated value of P^+ will be

$$af(O^+) = f(V, af(I^+)) \tag{6.1}$$

For example, this specialisation can occur when representing a set S as a list L. Assume the existence of a module providing the opaque type *set* and some basic operations on sets, including a procedure, *add*, for adding an element to a set (such a module can be found in Colvin et al. (2001)).

$$\mathit{add} \mathrel{\widehat{=}} (E, S, S') \textbf{ :- } \langle S' = \{E\} \cup S\rangle$$

To represent the set as a list we choose the coupling invariant to be the abstraction function $S = \text{ran}(L)$, where 'ran' returns the range or set of elements in a list. Using (6.1) we calculate the corresponding concrete procedure.

$$add^+ \mathrel{\widehat{=}} (E, L, L') \text{ :- } \langle \text{ran}(L') = \{E\} \cup \text{ran}(L) \rangle$$

This procedure outputs a list L' such that the elements of L' are the elements of L plus E. While this is valid, there are an infinite number of such lists because an element is not precluded from appearing multiple times in L'. Typically this will not be a practical implementation of the *add* procedure for lists.

Intuitively, however, since there is exactly one abstract output, there need only be one concrete output. In other words, the calculated value for P^+ should be of the form $O^+ = U$, for some term U. In fact, any term U with V and I^+ free that satisfies the following condition validates $O^+ = U$ as an implementation for P^+.

$$af(U) = f(V, af(I^+)) \tag{6.2}$$

This may be proved by substituting $O^+ = U$ for P^+ in Condition 4.6 and simplifying (strengthening).

In the set-as-list example, we require some value U for L' such that $\text{ran}(U) = \{E\} \cup \text{ran}(L)$. One obvious choice is $[E \mid L]$. Clearly, $\text{ran}([E \mid L]) = \{E\} \cup \text{ran}(L)$. Thus, we are free to implement the concrete version of *add* as $\langle L' = [E \mid L] \rangle$, which is a stronger constraint on L' than that calculated by (6.1). To formalise the choice for U we introduce demonic nondeterminism.

6.2 Demonic nondeterminism

In Hemer et al. (2002) a demonic choice operator ($\sqcap$) and its associated semantics and refinement laws are added to the refinement calculus. This allows the wide-spectrum language to express the *don't care* interpretation of nondeterminism using $\sqcap$, as well as the the default *don't know* interpretation of nondeterminism, within a single program. To understand the difference, consider the program $S \sqcap T$. It may be implemented by either of the programs S or T, as embodied in the following refinement laws:

$$S \sqcap T \sqsubseteq S$$
$$S \sqcap T \sqsubseteq T$$

Note the difference with program disjunction, where $S \vee T$ must be implemented by returning the answers for both S and T. For example, consider the program

$$\langle X = 0 \rangle \sqcap \langle X = 1 \rangle$$

This program is implemented by either the program $\langle X = 0 \rangle$ or the program $\langle X = 1 \rangle$. In contrast, the program $\langle X = 0 \rangle \vee \langle X = 1 \rangle$ is not; it must return both answers for X.

The identity of demonic choice is the program **magic**, that is, $(\textbf{magic} \sqcap S) = (S \sqcap \textbf{magic}) = S$. It is the (unimplementable) program that refines all other programs.

We may generalise the binary operator: given a program S with free variable X, the demonic choice between the set of programs formed by instantiating S with each possible value of X is given by

$$\sqcap X \bullet S(X)$$

This program is refined by $S(U)$, for all terms U. We may limit the range of X by introducing a *guard*. A guarded command $S \rightarrow T$ is **magic** if S fails, but behaves like S, T otherwise. To restrict the range of X to just those terms that satisfy some predicate P, we write

$$\sqcap X \bullet \langle P(X) \rangle \rightarrow S(X)$$

For example, a program that picks exactly one arbitrary element from a set A and sets some variable Y to have that value is:

$$\sqcap X \bullet \langle X \in A \rangle \rightarrow \langle Y = X \rangle$$

This is in contrast to the program $\langle Y \in A \rangle$, which binds Y to every element of A.

A generalised demonic choice over $P(X) \rightarrow S(X)$ is implemented by $S(U)$ for all terms U that satisfy $P(U)$. This is embodied in the following refinement law.

Law 8 (*Eliminate generalised demonic choice*)

$$\frac{P(U)}{(\sqcap X \bullet \langle P(X) \rangle \rightarrow S(X)) \sqsubseteq S(U)}$$

We may refine a program D to a generalised demonic choice if, for all terms X such that $P(X)$ holds, D is refined by $S(X)$. This is expressed by the following refinement law.

Law 9 (*Introduce generalised demonic choice*)

$$\frac{(\forall X \bullet P(X) \Rightarrow (D \sqsubseteq S(X))}{D \sqsubseteq (\sqcap X \bullet \langle P(X) \rangle \rightarrow S(X))}$$

6.3 Demonic choice in module calculation

When there is only one abstract output value for a procedure, i.e., when it is deterministic, we will typically want the corresponding concrete procedure to also be deterministic. In other words, when p is of the form $\{A\}, \langle O = f(V, I) \rangle$ for some assumption A and function f, the corresponding p^+ should be of the form $\{A^+\}, \langle O^+ = U \rangle$, where A^+ is the calculated assumption and U is some term involving V and I^+. However, when the coupling invariant allows many concrete representations of an abstract value, i.e., when the coupling invariant is an abstraction function of the form $I^+ = af(I)$, the applicable derivation specialisation (top left in Fig. 4) is not deterministic for O^+.

We solve this problem using demonic nondeterminism. Recall from Sect. 6.1 that $\langle O^+ = X \rangle$ is a valid implementation of p^+ for all terms X that satisfy (6.2). Expressing this formally:

$$(\forall X \bullet af(X) = f(V, af(I^+)) \Rightarrow (p^+(V, I^+, O^+) \sqsubseteq \{A^+\}, \langle O^+ = X \rangle))$$

From this, using Law 9 (*introduce generalised demonic choice*) we may deduce

$$p^+(V, I^+, O^+) \sqsubseteq (\sqcap X \bullet \langle af(X) = f(V, af(I^+))\rangle \rightarrow \{A^+\}, \langle O^+ = X\rangle) \qquad (6.3)$$

This specification allows more flexibility in the final implementation of the concrete procedure than the specification originally calculated (top left in Fig. 4). The implementor may choose any term U such that $af(U) = f(V, af(I^+))$, and from Law 8 (*eliminate generalised demonic choice*) the actual implementation of p^+ becomes $O^+ = U$. Without the reduction of nondeterminism, the implementor must retain each concrete value that corresponds to the abstract output.

In the set-as-list example, we would instantiate (6.3) to calculate the list implementation of add^+.

$$\sqcap X \bullet \langle \mathrm{ran}(X) = \{E\} \cup \mathrm{ran}(L)\rangle \rightarrow \langle L' = X\rangle$$

To refine this to code we choose some value for X that satisfies the guard. An obvious choice is $[E \mid L]$. It may be easily seen that $\mathrm{ran}([E \mid L]) = \{E\} \cup \mathrm{ran}(L)$, which is the proof obligation for applying Law 8 (*eliminate generalised demonic choice*). We can therefore implement add^+ as (E, L, L') **:-** $\langle L' = [E \mid L]\rangle$.

7 Related work

There is a large body of work on the deductive synthesis of logic programs, a survey of which appears in Basin et al. (2004). Deductive synthesis is a method for deriving a logic program from a specification, similar to the refinement calculus approach. A specification is manipulated using deduction rules (that are proved correct within the proof framework), until an executable program is reached. The various approaches to deductive synthesis vary mainly in their specification language; however, most use first-order logic since this can express both specifications and logic programming code. One of the most developed schemes for deductive synthesis is that of Lau and Ornaghi (1997b). They introduce a specification framework, which underlies the synthesis steps, providing axioms and derived relations.

The main difference between most deductive synthesis approaches and logic program refinement is the inclusion of assumptions in the wide-spectrum language. These act as preconditions, providing a context for refinement steps. Lau and Ornaghi (1997b) have a *conditional* specification, which includes an input relation for a procedure (e.g., types, modes) with respect to which the synthesis of the procedure can take place. The refinement calculus generalises this by allowing an assumption (input relation) for any arbitrary program fragment. A further difference is that in deductive synthesis the deduction rules are derived with the SLD computation rule in mind. Thus issues such as clause-ordering are dealt with during the synthesis process. The refinement approach defers such issues to a separate translation phase, where a particular implementation language (and computational model) are chosen and the wide-spectrum program is translated into code for that language. A translation scheme for Mercury programs (Somogyi et al. 1995) is described in Colvin et al. (2002).

Despite these differences, much of the work on logic program development in the

synthesis world should be applicable in the refinement calculus. The refinement calculus work has focused mainly on the process of developing logic programs, while much of the synthesis work has been developing strategies for deriving programs given particular forms of specification. We expect that such strategies can be formulated as sequences of refinement rules.

The examples in Sect. 3 draw on work on Prolog program transformations (Sterling and Shapiro 1994), in particular, transformations between the Prolog types *list* and *difference list* (Marriot and Søndergaard 1988). The relationship between the list and difference list implementations of *reverse* may also be defined with respect to higher-order program synthesis, as shown by Seres and Spivey (2000).

Specifications of procedures and modules in our wide-spectrum language (Sect. 4) are similar to Morgan's model-based module specifications for imperative programs (Morgan 1994), though in his case the modules provide a 'hidden' state (rather than type), which is not possible in traditional logic programs. Bancroft and Hayes (1993) have extended the imperative calculus to include module specifications with opaque types similar to ours. Our module specifications are similar to the module declarations of languages such as Mercury (Somogyi et al. 1995).

There are many other existing logic programming frameworks for modules or module-like encapsulation, e.g., (Srinivas and Jullig 1995; Lau and Ornaghi 1997a; Lau et al. 1999). Many of these define modules through the algebraic specification of abstract data types (ADTs) (Turski and Maibaum 1987). An implementation module may be derived by ensuring it maintains the axioms of the ADT. Read and Kazmierczak (1992) present a particular method of developing modular Prolog programs from axiomatic specifications. They write their programs in a module system based on that of extended ML. The specification of a module is written in the form of a set of axioms stating the required properties of the procedures of the module. To define the semantics of refinement, Prolog programs are considered to be equivalent to their predicate completions. The definition of module refinement in their approach is more general than the technique presented in this paper: any implementation that satisfies the axioms is valid (cf., interpretations between theories from logic (Turski and Maibaum 1987)). However, for modules with a large number of procedures, presenting an axiomatic specification of how the procedures interrelate is more problematic than with the model-based approach used in this paper. This is because axioms are required to define the possible interactions between procedures, whereas, in the approach used in this paper, each procedure is defined directly in terms of the model of the opaque type. In the algebraic approach, the proof of correctness amounts to showing that all the axioms of the specification hold for the implementation (Read and Kazmierczak 1992). For a module with a large number of procedures this can be quite complex. In comparison, the approach presented here breaks down the problem into data refinement of each procedure in isolation.

Imperative data refinement (Morgan 1994) has more similarities with our approach to module refinement. In that framework, a specification is augmented with the concrete variable and the coupling invariant, then refinement proceeds as normal, until the abstract variable is removed via diminution. Neither of the augment

and diminish steps are actual refinements, but as in our framework the resulting relationship between the abstract and concrete procedures is guaranteed to satisfy the conditions for data refinement.

The calculational method for deriving a concrete module from an abstract module and a coupling invariant in Sect. 5 is similar in style to that presented by Morgan and Gardiner (1990). The calculated concrete procedures can appear quite complex in both methods ((5.4) in this paper and Lemma 3 in (Morgan and Gardiner 1990)). However in the common situation in which the coupling invariant is an abstraction function, that is, the abstract value is a function of the concrete value, the one-point rules can be applied to simplify the calculated procedures to term replacements on the abstract procedure. These simplifications can occur in both settings. In either case, the bulk of the work revolves around eliminating the existentially quantified abstract state, and hence many data refinement techniques should be applicable in both settings. In the terminology of Morgan and Gardiner (1990), our calculated concrete procedure is *valid*, that is, it is a module refinement of the abstract. However, it is not *general* (unlike the imperative calculated concrete procedure), because there are other valid concrete procedures that are not (algorithmic) refinements of the calculated procedure. This necessitated the introduction of demonic nondeterminsm into the calculation process in Sect. 6.

8 Conclusions

This paper has described a cohesive framework for contextual refinement, module refinement, and the calculation of concrete modules. Contextual information simplifies the refinement process by allowing individual refinement steps and proof obligations to operate on the predicate level, with minimal reference to the structure of the program. Contextual information is collected via monotonicity laws, which not only simplifies proofs "by-hand", but can also be made transparent to the user when using a refinement tool (Hemer et al. 2001). The contextual laws presented in Sect. 3 have been used to develop a solution to the *N-queens* problem (Colvin 2002, Chapter 4), and also in the development of a term unification algorithm (Colvin et al. 2004). In this paper we make use of contextual information in providing laws for module refinement and calculation in a more convenient form.

Modules are an extension of the refinement calculus that allows data abstraction and encapsulation. In Sect. 4 we investigated an implementation of the module specifying a partial function type in Fig. 3. The partial function module has also been used in the development of a term unification algorithm (Colvin et al. 2004). The module calculation approach in Sect. 5 can be used to automatically derive a concrete module from an abstract module and coupling invariant. The calculated module is guaranteed to satisfy the conditions for module refinement, thus automatically discharging the proof obligations associated with module refinement. However, while the calculated module is a valid module refinement, there are in general many valid module refinements, some of which may be more efficient than the calculated version. This can occur in the common situation where the abstract procedure is deterministic and there are many possible concrete representations of

an abstract value. To overcome this problem, in Sect. 6 we introduced a demonic, or *don't care*, nondeterministic operator into the calculation process. This approach can be used to eliminate unwanted nondeterminism introduced by the coupling invariant.

Acknowledgments. The authors would like to thank David Hemer and three anonymous referees for constructive comments on earlier versions of the paper. The work reported in this paper was supported by Australian Research Council grant number A49937007: *Refinement Calculus for Logic Programming.*

References

Back, R. J. R. 1988. A calculus of refinements for program derivations. *Acta Informatica 25*, 593–624.

Bancroft, P. and Hayes, I. J. 1993. Refining a module with opaque types. In Proceedings, 16th Australian Computer Science Conference, Brisbane, G. Gupta, G. Mohay, and R. Topor, Eds. *Australian Computer Science Communications 15(1)*, 615–624.

Basin, D., Deville, Y., Flener, P., Hamfelt, A., and Nilsson, J. F. 2004. Synthesis of programs in computational logic. In *Program Development in Computational Logic: A Decade of Research Advances in Logic-Based Program Development*, M. Bruynooghe and K.-K. Lau, Eds. Lecture Notes in Computer Science, vol. 3049. Springer Verlag, 30–65.

Colvin, R. 2002. Contextual and data refinement for the refinement calculus for logic programs. Ph.D. thesis, School of Information Technology and Electrical Engineering, University of Queensland.

Colvin, R., Groves, L., Hayes, I. J., Hemer, D., Nickson, R., and Strooper, P. 2004. Developing logic programs from specifications using stepwise refinement. In *Program Development in Computational Logic: A Decade of Research Advances in Logic-Based Program Development*, M. Bruynooghe and K.-K. Lau, Eds. Lecture Notes in Computer Science, vol. 3049. Springer Verlag, 66–89.

Colvin, R., Hayes, I. J., Hemer, D., and Strooper, P. 2002. Translating refined logic programs to Mercury. In *Proceedings of the Twenty-Fifth Australasian Computer Science Conference*, M. Oudshoorn, Ed. Conferences in Research and Practice in Information Technology, vol. 4. Australian Computer Society, 33–40.

Colvin, R., Hayes, I. J., and Strooper, P. 1998. Data refining logic programs. In *International Refinement Workshop and Formal Methods Pacific 1998*, J. Grundy, M. Schwenke, and T. Vickers, Eds. Discrete Mathematics and Theoretical Computer Science. Springer-Verlag, 100–116.

Colvin, R., Hayes, I. J., and Strooper, P. 2000. Refining logic programs using types. In *Australasian Computer Science Conference (ACSC 2000)*, J. Edwards, Ed. IEEE Computer Society, 43–50.

Colvin, R., Hayes, I. J., and Strooper, P. 2001. A technique for modular logic program refinement. In *Logic Based Program Synthesis and Transformation (LOPSTR 2000), Selected Papers*, K.-K. Lau, Ed. LNCS, vol. 2402. Springer, 38–56.

Dijkstra, E. 1976. *A Discipline of Programming.* Prentice-Hall.

Hayes, I. J., Colvin, R., Hemer, D., Nickson, R., and Strooper, P. A. 2002. A refinement calculus for logic programs. *Theory and Practice of Logic Programming 2,* 4–5 (July–September), 425–460.

Hayes, I. J., Nickson, R., and Strooper, P. 1997. Refining specifications to logic

programs. In *Logic Program Synthesis and Transformation. Proc. of the 6th Int. Workshop, LOPSTR'96, Stockholm, Sweden, August 1996*, J. Gallagher, Ed. LNCS, vol. 1207. Springer, 1–19.

Hayes, I. J., Nickson, R., Strooper, P., and Colvin, R. 2000. A declarative semantics for logic program refinement. Tech. Rep. 00-30, Software Verification Research Centre, The University of Queensland.

Hemer, D., Colvin, R., Hayes, I. J., and Strooper, P. 2002. Don't care non-determinism in logic program refinement. In *Proceeding of Computing: the Australasian Theory Symposium*, J. Harland, Ed. Electronic Notes in Computer Science (ENTCS), vol. 61. Elsevier Science. http://www.elsevier.nl/locate/entcs/volume61.html.

Hemer, D., Hayes, I. J., and Strooper, P. 2001. Refinement Calculus for Logic Programming in Isabelle/HOL. In *Theorem Proving in Higher Order Logics, 14th International Conference, TPHOLs 2001*, R. Boulton and P. Jackson, Eds. LNCS, vol. 2152. Springer, 249–264.

Hill, P. and Lloyd, J. 1994. *The Gödel Programming Language.* MIT Press.

Lau, K.-K. and Ornaghi, M. 1997a. Forms of logic specifications: A preliminary study. In *Logic Program Synthesis and Transformation. Proc. of the 6th Int. Workshop, LOPSTR'96, Stockholm, Sweden, August 1996*, J. Gallagher, Ed. LNCS, vol. 1207. Springer, 295–312.

Lau, K.-K. and Ornaghi, M. 1997b. The relationship between logic programs and specifications — the subset example revisited. *J. Logic Programming 30,* 3 (March), 239–257.

Lau, K.-K., Ornaghi, M., and Tärnlund, S.-Å. 1999. Steadfast logic programs. *J. Logic Programming 38,* 3 (March), 259–294.

Liskov, B. and Guttag, J. 1986. *Abstraction and Specification in Program Development.* MIT Press.

Marriot, K. and Søndergaard, H. 1988. Prolog program transformation by introduction of difference-lists. In *In Proceedings International Computer Science Conference '88.* IEEE Computer Society, 206–213.

Morgan, C. 1994. *Programming from Specifications*, Second ed. Prentice Hall.

Morgan, C. and Gardiner, P. 1990. Data refinement by calculation. *Acta Informatica 27*, 481–503.

Nickson, R. and Hayes, I. J. 1997. Supporting contexts in program refinement. *Science of Computer Programming 29,* 3, 279–302.

Partsch, H. A. 1990. *Specification and Transformation of Programs.* Springer-Verlag.

Read, M. and Kazmierczak, E. 1992. Formal program development in modular Prolog. In *Logic Program Synthesis and Transformation. Proc. of LOPSTR'91, University of Manchester, July 1991*, T. Clement and K.-K. Lau, Eds. LNCS, vol. 1207. Springer-Verlag, 69–93.

Seres, S. and Spivey, M. 2000. Higher-order transformation of logic programs. In *Proceedings of the Tenth International Workshop on Logic-based Program Synthesis and Transformation (LOPSTR 2000)*, K.-K. Lau, Ed. LNCS, vol. 2042. Springer-Verlag, 57–68.

Shapiro, E. 1989. The family of concurrent logic programming languages. *ACM Comput. Surv. 21,* 3, 413–510.

Somogyi, Z., Henderson, F., and Conway, T. 1995. Mercury, an efficient purely declarative logic programming language. In *Proceedings of the Eighteenth Australasian Computer Science Conference*, R. Kotagiri, Ed. Australian Computer Science Communications, Glenelg, South Australia, 499–512.

Srinivas, Y. V. and Jullig, R. 1995. Specware: Formal support for composing software. In *Mathematics of Program Construction*, B. Möller, Ed. Lecture Notes in Computer Science, vol. 947. Springer, 399–422.

Sterling, L. and Shapiro, E. 1986. Second edition 1994. *The Art of Prolog*. MIT Press.

Turski, W. M. and Maibaum, T. S. E. 1987. *Specification of Computer Programs*. Addison-Wesley.